Learnability of Subsequence Languages

AUTHOR(S):

MATSUMOTO, Satoshi; SHINOHARA, Ayumi

CITATION:

MATSUMOTO, Satoshi ...[et al]. Learnability of Subsequence Languages. 数理解析研究所講究録 1996, 950: 250-256

ISSUE DATE:

1996-05

URL:

http://hdl.handle.net/2433/60311

数理解析研究所講究録
950巻 1996年 250-256

Learnability of Subsequence Languages

松本 哲志* 篠原 歩
Satoshi MATSUMOTO Ayumi SHINOHARA
{matumoto, ayumi}@rifis.kyushu-u.ac.jp
Research Institute of Fundamental Information Science,
Kyushu University 33, Fukuoka 812, JAPAN.

Abstract

For a string w, we define the subsequence language $L(w)$ by the set of all strings which contains w as a subsequence. We denote the class of subsequence languages by $\mathcal{S}$. Let $\mathcal{S}^k$ denote the class of unions of at most k subsequence languages, and $\mathcal{S}^* = \bigcup_{k\geq 0}\mathcal{S}^k$. It it known that $\mathcal{S}$ is not polynomial-time PAC learnable unless **RP**=**NP**. One approach to overcome this computational hardness is to relax the problem by allowing a learning algorithm to make membership queries and equivalence queries. We show that the class $\mathcal{S}$ is exactly learnable using membership queries only. The class $\mathcal{S}^*$ is exactly learnable using equivalence and membership queries. As negative results, we show that the class $\mathcal{S}^k$ is not polynomial-time PAC learnable unless **RP**=**NP** and learning $\mathcal{S}^*$ is as hard as learning DNF.

1 Introduction

For a string w, we define the subsequence language $L(w)$ by the set of all strings which contains w as a subsequence. We denote by $\mathcal{S}$ the class of all subsequence languages. For $k \geq 1$, let $\mathcal{S}^k$ denote the class of unions of at most k subsequence languages, and $\mathcal{S}^* = \bigcup_{k\geq 0}\mathcal{S}^k$.

It is known that the consistency subsequence problem for $\mathcal{S}$ is an **NP**-complete problem[10]. Therefore, the class $\mathcal{S}$ is not polynomial-time learnable in the framework of PAC learning model [14] under the assumption of **NP**≠**RP**[11].

One approach to overcome this computational hardness is to relax the problem by allowing a learning algorithm to make membership queries and equivalence queries [2, 3, 5, 6]. By using membership and equivalence queries, the learning algorithm can actively collect information on a target concept.

In Section 3, we develop learning algorithms for $\mathcal{S}$ and $\mathcal{S}^*$ using queries. At first, we show that the class $\mathcal{S}$ is learnable by membership queries only. Our algorithm runs in $O(|s|^2)$ time using $O(|s|)$ membership queries, where s is the target string to be identified. Next, we show a learning algorithm for $\mathcal{S}^*$ using both membership and equivalence queries. Its running time is $O(mn^2 + m^2n)$, using $n+1$ equivalence queries and at most mn membership queries, where n is the size of the target set to be identified and m is the length of the longest counterexample seen so far.

In section 4, we show that learning $\mathcal{S}^*$ is as hard as learning general Boolean formulae in disjunctive normal form (DNF), by showing a prediction preserving reduction [13] from DNF to $\mathcal{S}^*$. The class of DNF is widely believed not to be polynomial-time learnable [1, 4, 7] in PAC learning model. In this sense, the class $\mathcal{S}^*$ is unlikely to be polynomial-time learnable in PAC learning model. We also prove that the consistency problem for $\mathcal{S}^k$ is **NP**-complete for any fixed $k \geq 1$. Thus the class $\mathcal{S}^k$ is also not polynomial-time learnable in PAC learning model under the assumption of **NP**≠**RP**.

*This author is a Research Fellow of the Japan Society for the Promotion of Science (JSPS). The author's research is partly supported by Grants-in-Aid for JSPS research fellows from the Ministry of Education, Science and Culture, Japan.

2 Preliminaries

Let Σ be a finite alphabet. We denote by Σ^* the set of all strings. The *empty string* is denoted by ε. For a string $u \in \Sigma^*$, the length of u is denoted by $|u|$. Let $u[i]$ denote the i-th symbol of u, and $u[\tilde{i}]$ the string which is obtained by eliminating $u[i]$ from u. We say that a string w is a *subsequence* of u if w can be obtained from u by deleting zero or more symbols from it. We write $w \leq u$ if w is a subsequence of u. We also say that u is a *supersequence* of w, and write $u \geq w$. In particular, We write $w < u$ if $w \leq u$ and $w \neq u$. The *subsequence language of* w, denoted by $L(w)$, is the set of all supersequences of w, that is, $L(w) = \{u \in \Sigma^* \mid u \geq w\}$. We say that w is the *base* for $L(w)$. For a set H of strings , we define $L(H) = \bigcup_{w \in H} L(w)$, and $L(H)$ is called the *subsequence language defined by* H. The *size* of H, denoted by $|H|$, is the number of strings in H. We define $\mathcal{S} = \{S \subseteq \Sigma^* \mid |S| = 1\}$. For $k \geq 1$, $\mathcal{S}^k = \{S \subseteq \Sigma^* \mid |S| \leq k\}$ and $\mathcal{S}^* = \bigcup_{k \geq 1} \mathcal{S}^k$. Note that $\mathcal{S}^1 = \mathcal{S} \cup \{\phi\}$. When it is clear from the context, the notation $\mathcal{S}^k$ is abused to stand for the class of languages $\{L(H) \mid H \in \mathcal{S}^k\}$, and so are $\mathcal{S}^*$ and $\mathcal{S}$.

In what follows, let $H_* \in \mathcal{S}^*$ to be identified, and we say that the set H_* is a *target*. We assume that H_* contains no redundant string, that is, $L(H_* - \{s\}) \neq L(H_*)$ for any $s \in H_*$. A string w is called a *positive example* of $L(H_*)$ if $w \in L(H_*)$, and a *negative example* otherwise.

First we introduce exact learning model via queries due to Angluin [3]. In this model, learning algorithms can access to *oracles* that will answer specific kinds of queries about the unknown set $L(H_*)$. We consider the following two oracles.

1. *Membership oracle* Mem_{H_*}: The input is a string w. The output is "*yes*" if $w \in L(H_*)$ and "*no*" if $w \notin L(H_*)$. The query is called a *membership query*.
2. *Equivalence oracle* $Equiv_{H_*}$: The input is a finite set H of strings. If $L(H) = L(H_*)$, the output is "*yes*". Otherwise, it returns a *counterexample* $w \in L(H) \cup L(H_*) - L(H) \cap L(H_*)$. The query is called an *equivalence query*.

A *learning algorithm* $\mathcal{A}$ may collect information about H_* using membership and equivalence queries. The goal of the learning algorithm $\mathcal{A}$ is exact identification in polynomial time, that is, $\mathcal{A}$ must halt and output a set $H \in \mathcal{S}^*$ such that $L(H) = L(H_*)$, furthermore the running time of $\mathcal{A}$ is a polynomial in the size of H_* and the length of counterexample returned by the equivalence oracle so far.

3 Learning algorithms using queries

In learning a subsequence language defined by H_*, a positive example $w \in L(H_*)$ gives crucial information to a learner $\mathcal{A}$. Since the positive example w is a supersequence of some base $s \in H_*$, the learner $\mathcal{A}$ can identify s by simply deleting redundant symbols from w with asking to the membership oracle for $L(H_*)$.

The next Lemma 1 supports the correctness of the following idea to find a base in H_* from a positive example $w \in L(H_*)$.

while ($w[\tilde{i}] \in L(H_*)$ for some i) **do**
  /* Since $w[i]$ is redundant, delete it */
  $w := w[\tilde{i}]$;
return w

Lemma 1. Let $H \in \mathcal{S}^*$ and $w \in L(H)$. If $w[\tilde{i}] \notin L(H)$ for all i, then $w \in H$.

Proof Since $w \in L(H)$, there exists $s \in H$ such that $w \in L(s)$. Suppose $w \neq s$, that is, $s < w$. Then, there exists i with $s \leq w[\tilde{i}]$, which implies $w[\tilde{i}] \in L(s) \subseteq L(H)$. This contradicts with the assumption that $w[\tilde{i}] \notin L(H)$ for all i. Therefore, $w = s \in H$. □

Moreover, Lemma 2 guarantees that for each symbol $w[i]$ in the given positive example w, it is enough to check whether $w[i]$ is redundant or not only once.

Procedure: *Shrink*(w)
Input: $w \in L(H_*)$. /* positive example of $L(H_*)$ */
Given: Mem_{H_*}.
Output: $v \in H_*$.
begin
  $v := w$;
  for $i = |w|$ **downto** 1 **do**
    if $Mem_{H_*}(v[\tilde{i}]) =$ "*yes*" **then**
      $v := v[\tilde{i}]$;
  return v;
end

Figure 1: procedure *Shrink*

Lemma 2. Let $H \in \mathcal{S}^*$ and $x, y \in \Sigma^*$. If $xy \notin L(H)$, then $x'y' \notin L(H)$ for any $x' \leq x$ and $y' \leq y$.

Proof Suppose that there exists strings $x' \leq x$ and $y' \leq y$ such that $x'y' \in L(H)$. Since $x'y' \leq xy$, we have $xy \in L(H)$, which is a contradiction. □

By these arguments, we can construct the procedure *Shrink* in Figure 1 which finds a base in H_* from a positive example.

Theorem 1. If the algorithm *Shrink* gets a positive example $w \in L(H_*)$ as an input, *Shrink* runs in $O(|w|^2)$ time and outputs a base $v \in H_*$ using $|w|$ membership queries.

Proof The running time and the number of queries are obvious. According to the correctness of the algorithm, by Lemma 2, we can show that $v[\tilde{j}] \notin L(H_*)$ for all j, which ensures that $v \in H_*$ by Lemma 1. □

By Theorem 1, whenever a learning algorithm $\mathcal{A}$ succeeds to get a positive example $w \in L(H_*)$, $\mathcal{A}$ can identify a base $s \in H_*$ by the procedure *Shrink*(w). Especially, if H_* consists of only one string s, one positive example is enough to identify H_*. In fact, as we will show, $\mathcal{A}$ can generate a positive example by using the membership queries. The idea is based on the following simple observation.

Observation 1. For an alphabet $\Sigma = \{a_1, a_2, \cdots, a_d\}$, let ${p_\Sigma}^r$ be the string $(a_1 a_2 \cdots a_d)^r$, the r-times repetition of the string $a_1 a_2 \cdots a_d$. Then any string of length r is a subsequence of ${p_\Sigma}^r$.

We show our learning algorithm $LEARN_1$ in Figure 2 which works when the target H_* is known to be a singleton $\{s\}$.

Theorem 2. The learning algorithm $LEARN_1$ exactly identifies every set $\{s\} \in \mathcal{S}$ in $O(l^2)$ time using at most $dl + l + 1$ membership queries only, where $l = |s|$ and $d = |\Sigma|$.

Proof Since $s \leq {p_\Sigma}^i$ for some $i \leq l$, $LEARN_1$ can get a positive example ${p_\Sigma}^i$ of $L(\{s\})$ using at most $l + 1$ membership queries. Since $|{p_\Sigma}^i| = di \leq dl$, the procedure *Shrink*$({p_\Sigma}^i)$ asks at most dl membership queries. The running time is obviously $O(l^2)$. □

Next, we show our learning algorithm $LEARN_*$ for general case $|H_*| \geq 0$ which uses both equivalence and membership queries.

Procedure: $LEARN_1$
Given: Mem_{H_*}.
Output: the base $v = s$.
begin
  $i := 0$;
  while $Mem_{H_*}(p_\Sigma{}^i) \neq$ "*yes*" **do**
    $i := i + 1$;
  $v :=$ *Shrink*$(p_\Sigma{}^i)$;
  output v;
end

Figure 2: A learning algorithm using queries for the case $H_* = \{s\}$

Theorem 3. The algorithm $LEARN_*$ exactly identifies every set $H_* \in \mathcal{S}^*$ in $O(mn^2 + m^2n)$ time using $n+1$ equivalence queries and at most mn membership queries, where $n = |H_*|$ and m is the length of the longest counterexample seen so far.

Proof By induction, we can show that the set H in the algorithm $LEARN_*$ is always a subset of H_*. Since the counterexample p is in $L(H_*) - L(H)$, the string *Shrink*(p) is in $H_* - H$ by Theorem 1. Thus, the **while** loop is repeated n times and $LEARN_*$ outputs H_*. □

4 Hardness Results on the PAC-Learnability

In the previous section, we have shown that the class $\mathcal{S}$ can be identified by membership queries only, and the class $\mathcal{S}^*$ by membership and equivalence queries. In this section, we show that these classes are hard to learn without membership queries. In the PAC-learning model [14], we will prove that the class $\mathcal{S}^k$ is not polynomial-time learnable for any fixed $k \geq 1$ unless **NP=RP**. We also show that learning $\mathcal{S}^*$ is as hard as learning the class of general DNF formulae, which has been believed not to be polynomial-time learnable [1, 4, 7].

Theorem 4. If the class $\mathcal{S}^*$ is polynomial-time learnable, so is the class of general DNF.

Proof We will show that the prediction preserve reduction [13] from the class of DNF to $\mathcal{S}^*$.

Each DNF formula ψ corresponds to the set H_ψ of strings as follows: Without loss of generality, we can assume that no term in ψ contains both positive literal x_i and negative literal $\neg x_i$ simultaneously. For each term T in ψ, the set H_ψ contains the string $\pi_T = a_1 \# a_2 \# \cdots \# a_n$, where

$$a_i = \begin{cases} 1 & \text{if } x_i \text{ appears in } T, \\ 0 & \text{if } \neg x_i \text{ appears in } T, \\ \varepsilon & \text{otherwise.} \end{cases}$$

We define a mapping g which transforms a truth assignment $w = b_1 b_2 \cdots b_n \in \{0,1\}^n$ for ψ to the string $g(w) \in \{0,1,\#\}^{2n-1}$ such that w satisfies ψ iff $g(w)$ is in $L(H_\psi)$ as follows:

$$g(b_1 b_2 \cdots b_n) = b_1 \# b_2 \# \cdots \# b_{n-1} \# b_n, \qquad b_i \in \{0,1\}.$$

Obviously, g can be computed in polynomial time.

Procedure: $LEARN_*$
Given: Mem_{H_*} and $Equiv_{H_*}$.
Output: the target set $H = H_*$.
begin
  $H := \phi$;
  while $Equiv_{H_*}(H) \neq$ "*yes*" **do**
    begin
      let p be the counterexample which $Equiv_{H_*}(H)$ returns;
      $s := Shrink(p)$;
      $H := H \cup \{s\}$;
    end
  output H;
end

Figure 3: A learning algorithm using queries for general case $|H_*| \geq 0$

First we show that if a truth assignment $w \in \{0,1\}^n$ satisfies ψ then $g(w) = b_1\#b_2\#\cdots\#b_n \in L(H_\psi)$. Since w satisfies ψ, there exists a term T in ψ which is satisfied by w. For the corresponding string $\pi_T = a_1\#a_2\#\cdots\#a_n$, we prove that $a_i = \varepsilon$ or $a_i = b_i$ for each i, which implies that $g(w)$ is a supersequence of π_T. We have only to consider the case $a_i \neq \varepsilon$. If $a_i = 1$, the positive literal x_i appears in T. Then $b_i = 1$ since w satisfies T. If $a_i = 0$, the negative literal $\neg x_i$ appears in T. Then $b_i = 0$ since w satisfies T. Thus $g(w)$ is a supersequence of π_T. Therefore $g(w) \in L(\pi_T) \subseteq L(H_\psi)$.

We show the converse. Assume that $g(w) = b_1\#b_2\#\cdots\#b_n \in L(H_\psi)$. There exists a string $\pi_T = a_1\#a_2\#\cdots\#a_n$ in H_ψ which is a subsequence of $g(w)$. Because of the delimiter symbol $\#$, if $a_i \neq \varepsilon$ then $a_i = b_i$. For a positive literal x_i in T, $b_i = 1$ since $a_i = 1$. If the negative literal $\neg x_i$ is in T, $b_i = 0$ since $a_i = 0$. Thus w satisfies T. Therefore ψ is satisfied by w. □

Let Y and N be mutually disjoint nonempty finite sets of stings. A set H of strings is *consistent with* $\langle Y, N\rangle$ if $Y \subseteq L(H)$ and $N \cap L(H) = \phi$. The *consistency problem for* $\mathcal{S}^k$ is a problem to decide whether there exists a set $H \in \mathcal{S}^k$ consistent with $\langle Y, N\rangle$.

Theorem 5. The consistency problem for $\mathcal{S}^k$ is **NP**-complete for any $k \geq 1$.

Proof The case of $k = 1$ is shown in [8, 9, 10]. Here we prove the **NP**-completeness for $k \geq 2$. Obviously the problem is in **NP**. We show the reduction from the consistency problem for k-term DNF, which is defined as follows: Given a pair $\langle Y, N\rangle$ of sets of assignments, decide whether or not there exists a k-term DNF which is satisfied by all assignments in Y but is not satisfied by any assignment in N. The problem is known to be **NP**-complete [12]. The basic idea of the reduction is similar to the proof of Theorem 4. Let Y and N be mutually disjoint sets of assignments over variables $\{x_1, x_2, \cdots, x_n\}$ in an arbitrary instance of the consistency problem for k-term DNF. Using Y and N, we define Y' and N' as follows:

$$\begin{aligned} Y' &= \{g(w) \mid w \in Y\}, \\ N' &= \{g(w) \mid w \in N\} \cup \{(0101\#)^{n-2}0101\}, \end{aligned}$$

where the mapping g is defined in Theorem 4.

We can show that the existence of k-term DNF formula ψ consistent with $\langle Y, N\rangle$ implies the existence of a set H of at most k strings consistent with $\langle Y', N'\rangle$ in the same way to the proof of Theorem 4.

Now we show the converse. Let H be a set of at most k strings consistent with $\langle Y', N'\rangle$. Without loss of generality, H contains no redundant strings, that is, for any $s \in H$, $Y' \not\subseteq L(H - \{s\})$. Let s be a string in H. The string s is a subsequence of some positive example $g(w) = b_1\#b_2\#\cdots\#b_n \in Y'$ since H contains no redundant strings. If the number of #'s in s is less than $n-1$, the negative string $(0101\#)^{n-2}0101$ becomes a supersequence of s, which is a contradiction. Thus the string s must be of the form $a_1\#a_2\#\cdots\#a_n$, where $a_i \in \{0, 1, \varepsilon\}$.

Now we define the k-term DNF formula ψ_H corresponding to the set H. For each string $s \in H$, the formula ψ_H contains the term $T_s = l_1 \cdots l_n$, where

$$l_i = \begin{cases} x_i & \text{if } a_i = 1, \\ \neg x_i & \text{if } a_i = 0, \\ 1 & a_i = \varepsilon. \end{cases}$$

We can show that w satisfies ψ_H if and only if $g(w) \in L(H)$ for any $w \in \{0,1\}^n$ in the same way in the proof of Theorem 4. □

We get the following corollary by the results due to Pitt and Valiant [12].

Corollary 1. The class $\mathcal{S}^k$ is not polynomial-time learnable for any $k \geq 1$ unless **RP=NP**.

5 Conclusion

We have discussed the learnability of $\mathcal{S}^*$ in two learning models. The class $\mathcal{S}^k$ is not polynomial-time learnable for any fixed $k \geq 1$ unless **RP=NP** in PAC learning model. The class $\mathcal{S}^*$ is believed not to be polynomial-time learnable since learning $\mathcal{S}^*$ is as hard as learning the class of DNF. We have also shown that the class $\mathcal{S}^*$ is exactly learnable using membership and equivalence queries. In particular, the class $\mathcal{S}$ is exactly learnable using membership queries only. We summarize these results in Figure 4.

As future works, we will study the learnabilities of intersection of subsequence languages. It may be interesting question whether the class $\mathcal{S}^*$ can be identified by membership queries only.

	PAC learning model	exact learning via queries		
			membership	equivalence
$\mathcal{S}$	*No* (**RP≠NP**) [10]	*Yes*	$dl + l + 1$	0
$\mathcal{S}^k$	*No* (**RP≠NP**)	*Yes*	km	$k + 1$
$\mathcal{S}^*$	as hard as DNF	*Yes*	mn	$n + 1$

Figure 4: Summary of the results in Section 3 and 4. In the table, l is the length of the string to be identified, m is the length of the counterexample seen so far, n is the size of the target and $d = |\Sigma|$.

6 Acknowledgments

The authors wish to acknowledge Professor Takeshi Shinohara, Hiroki Ishizaka and Hiroki Arimura for their helpful suggestions and encouragement.

References

[1] H. Aizenstein and L. Pitt. On the learnability of disjunctive normal from formulas. *Machine Learning*, 19:183–208, 1995.

[2] D. Angluin. Learning regular sets from queries and counterexamples. *Information and Computation*, 75:87–106, 1987.

[3] D. Angluin. Queries and concept learning. *Machine Learning*, 2:319–342, 1988.

[4] D. Angluin and M. Kharitonov. When won't membership queries help. *Journal of Computer and System Sciences*, 50:336–355, 1995.

[5] H. Arimura, H. Ishizaka and T. Shinohara. Learning unions of tree patterns using queries. In *Proceedings of 6th Workshop on Algorithmic Learning Theory*, pages 66–79, 1995.

[6] H. Ishizaka, H. Arimura and T. Shinohara. Finding tree patterns consistent with positive and negative examples using queries. In *Proceedings of 5th Workshop on Algorithmic Learning Theory*, pages 317–332, 1994.

[7] M. J. Kearns and L. G. Valiant. Cryptographic limitations on learning boolean formulae and finite automata. *Journal of the ACM*, 41:67–95, 1994.

[8] M. Middendorf. The shortest common nonsubsequence problem is **NP**-complete. *Theoretical Computer Science*, 108:365–369, 1993.

[9] M. Middendorf. On finding minimal, maximal and consistent sequences over a binary alphabet. *Theoretical Computer Science*, 145:317–327, 1995.

[10] S. Miyano, A. Shinohara and T. Shinohara. Which classes of elementary formal systems are polynomial-time learnable? In *Proceedings of the Workshop on Algorithmic Learning Theory*, pages 139–150, 1991.

[11] B. K. Natarajan. *Machine Learning a theoretical approach.* Morgan Kaufmann, 1991.

[12] L. Pitt and L. G. Valiant. Computational limitations on learning from examples. *Journal of the ACM*, 35:965–984, 1988.

[13] L. Pitt and M. K. Warmuth. Prediction-preserving reducibility. *Journal of Computer and System Sciences*, 41:430–467, 1990.

[14] L. G. Valiant. A theory of the learnable. *Communications of the ACM*, 27:1134–1142, 1984.